REMARKS/ARGUMENTS

Claims 1-19 are pending and rejected.

Claims 1-5 and 8-14 are rejected under 35 U.S.C. § 102(e) as being taught by Kyker et al., (hereinafter "Kyker"), U.S. Patent Number 6,578,138. Claims 6-7 and 15-19 are rejected under 35 U.S.C. § 103(a) as being unpatentable over Kyker and further in view of Rotenberg et al., (hereinafter "Rotenberg"), "A Trace Cache Microarchitecture and Evaluation" IEEE ©1999. Claims 1, 3-8, 12-15 and 17-19 are rejected under § 103(a) as being unpatentable over Patel et al., *Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing* (hereinafter "Patel"), in further view of Johnson, U.S. Patent No. 5,924,092 (hereinafter "Johnson"). Claims 2, 9-11 and 16 are rejected under 35 U.S.C. § 103(a) as being unpatentable over Patel, in view of Johnson, in further view of Peled et al., U.S. Patent No. 6,076,144 (hereinafter "Peled").

A. § 102 rejections under Kyker

Applicants submit the cited references do not teach, suggest or disclose at least "[a] cache comprising: a cache line to store an instruction segment further comprising a plurality of instructions stored in sequential positions of the cache line in reverse program order" (*e.g.*, as described in claim 1).

The Office Action asserts Kyker teaches the relevant limitations at column 2, line 59 – column 3, line 33, including Figure 1 and Figure 2. *See* Office Action dated 6/12/2008, paragraph 5. Applicants disagree.

The first paragraph of the cited section, column 2, line 59 – column 3, line 33, states:

> FIGS. 1 and 2 illustrates an exemplary embodiment of a method of unrolling loops within a trace cache according to the present invention. In particular, this first exemplary method is applied to two exemplary traces T1 and T2 (illustrated in FIG. 2), which will be utilized throughout the description for convenience. Traces T1 and T2 represent traces built according to known methods described above, and will be contrasted with traces T1' and T2', which are built according to exemplary embodiments of methods according to the present invention. Exemplary trace T1 includes a total of four micro-ops, three of which form a loop. Trace T1 first includes a trace head T.sub.H, followed by the head of the loop L.sub.H. The third micro-op of trace T1 is the second micro-op of the loop, L.sub.2, and the fourth micro-op of the trace is the third micro-op of the loop, L.sub.3. The final micro-op L.sub.3 includes, for example, the backward taken branch whose target address is the head of the loop L.sub.H. The second exemplary trace T2 includes the same loop, L.sub.H, L.sub.1, L.sub.2, but does not include any micro-op preceding the loop itself. Accordingly, L.sub.H is also the trace head (and is therefore illustrated as T.sub.H /L.sub.H), followed by L.sub.2 and L.sub.3 (see FIG. 2).

The first paragraph of the cited section describes the "unrolling" of the illustrated traces T1 and T2 of Figures 1 and 2. Trace T1 includes four elements – a trace head T.sub.H, and three elements that form the "loop": L.sub.H. L.sub.2, and L.sub.3. The second trace T2 includes the same loop, L.sub.H, L.sub.1, L.sub.2, but does not include any micro-op preceding the loop itself. In essence, the first paragraph describes the content and structure of the traces T1 and T2 to be "unrolled", but does not describe the unrolling process itself.

The second paragraph of the cited section states:

In the first exemplary method, the trace cache does not end the current trace (*i.e.*, the trace being built) at the occurrence of the backward taken branch, as may be done in a conventional trace cache. Rather, when the trace cache determines that a loop is present, the trace cache continues to build the trace by building additional iterations of the loop until the trace is a

minimal length, for example until the trace is greater than twelve micro-ops long. In other words, the trace cache builds the loop repeatedly until the trace is, for example, over two trace lines long. In an exemplary embodiment, the existence of a loop is established by checking for the existence of a backward-taken branch. It should be understood, however, that the determination that a loop exists need not be an active determination. Rather, the phrasing is simply used to designate a general situation in which a loop exists. If no loop exists for a given micro-op, the trace cache may continue to build the trace as usual, *i.e.*, according to given parameters and limitations (such as length, etc.) which may be incorporated in a trace cache which does not unroll loops according to the present invention.

The second paragraph of the cited section describes the "unrolling" process of Kyker in that when the trace cache determines that a loop is present, the trace cache continues to build the trace by building additional iterations of the loop until the trace is a minimal length. In short, the trace cache *builds a loop.* The cited section further describes that the trace cache *builds a loop* by checking for the existence of a backward taken branch. Kyker defines a backward taken branch as follows: "[a] backward taken branch generally occurs when the target address of a branch is a prior micro-op, and in particular, for purposes of this description, a prior micro-op of the trace." *See* column 1, lines 30-33. Therefore, while building a loop, a backward taken branch may be utilized by determining that the target address of the following branch is a prior micro-op, *thereby enabling the building of the loop.*

Moreover, contrary to the Office Action's assertions, determining that the target address of the following branch is a prior micro-op to enable the building of a loop is not the same as executable instructions stored in reverse program order. The Office Action asserts when L_H is

encountered, previously stored instructions L_2 and L_3 are stored again in consecutive lines in the cache, further alleging that the trace cache reverses the program direction and stores previously stored instructions again in the trace cache when a loop is encountered. *See* Office Action dated 6/12/2008, paragraph 64. Respectfully, Applicants submit the Office Action misses the point. First, merely jumping backwards to a previous instruction is not reversing program direction, it is merely jumping backwards to a previous instruction (that may be for the sake of any number of reasons – error correction, execution stall) is not reversing program direction. The "direction" of execution remains the same; the point at which the execution process is at is merely interrupted, reset, and restarted. Moreover, even if the Office Action's assertion regarding reversing program direction were regarded to be true (they are not), changing program execution "direction" by jumping back to a previous instruction as the Office Action describes is not the same as executing instructions in reverse program order.

Therefore, Applicants submit the cited section does not teach or suggest a cache comprising a cache line to store an instruction segment further comprising a plurality of instructions stored in sequential positions of the cache line in reverse program order (*e.g.*, as described in claim 1). Indeed, reverse program order is not discussed at all, even generally. One of ordinary skill in the art will readily understand that a trace cache that builds a loop is not the same as a cache that stores a plurality of instructions stored in sequential positions of the cache line in reverse program order. Applicants submit the Kyker reference fails to teach or suggest at least the above-discussed limitations of claim 1, and therefore the current § 102 rejection of claim 1 is lacking and should be withdrawn.

The Office Action requests clarification regarding the limitation "reverse program order". *See id*. Clarification may be found at least at page 5, line 21 – page 6, line 12 of the specification. Applicants reiterate, as evident from the specification, one of ordinary skill in the art will readily understand that a trace cache that builds a loop is not the same as a cache that stores a plurality of instructions stored in sequential positions of the cache line in reverse program order.

Rotenberg fails to make up for the deficiencies of Kyker. Rotenberg is directed to trace cache sequencing, selection, and prediction. It does not teach at least "[a] cache comprising: a cache line to store an instruction segment further comprising a plurality of instructions stored in sequential positions of the cache line in reverse program order" (*e.g.*, as described in claim 1).

B. § 103 rejections under Patel and Johnson

Applicants submit the cited references do not teach, suggest or disclose at least "[a] cache comprising: a cache line to store an instruction segment further comprising a plurality of instructions stored in sequential positions of the cache line in reverse program order" (*e.g.*, as described in claim 1).

First, Applicants agree with the Office Action's assertion Patel has not explicitly taught storing the plurality of instructions in sequential positions of cache line in reverse program order. *See* Office Action dated 6/12/2008, paragraph 29.

Patel discloses the improvement of fetch rates in trace caches by employing branch promotion and trace packing. Branch promotion removes the overhead resulting from dynamic branch prediction by applying static branch prediction to strongly biased branches. Trace

packing packs as many instructions as possible into a pending trace so that more instructions segments may be fetched during a single fetch cycle. However, Patel neither teaches nor suggests that the fetch rates may be improved by reversing the order of the instructions in the traces. Applicants submit there would be no motivation to do so, since reversing instruction order would not appear to improve fetch rates given the teaching of Patel.

By definition, a trace is a sequence of dynamically executed instructions, which may originally reside in non-continuous portions of the program memory, starting with a single entry instruction and ending with multiple exit instructions. For a typical trace, the head of the trace, *i.e.*, the first instruction in a sequence, is followed by the next executable instruction in the sequence, then the next, and so on. If the Office Action's assertions (discussed above) are correct, the first instruction is accessed and modified more than the second, third, etc., instructions. This is contrary to the known operation of the typical trace.

Applicants submit it is unclear how accessing the first instruction in a trace more frequently than the second instruction, and accessing the second instruction more frequently than the third, and so on, would improve the performance of a trace (thereby eliminating any motivation to do so). Since the trace defines sequential instructions, which perform a particular operation, accessing the instructions in decreasing frequency would in no way advance the completion of the particular operation. Indeed, such access of the trace would hinder the completion, thereby defeating the purpose of the trace.

Moreover, it is unclear how modifying the first instruction in a trace more frequently than the second instruction, and modifying the second instruction more frequently than the third, and so on, would improve the performance of a trace. Again, since the trace defines sequential

instructions, which perform a particular operation, modifying the instructions in decreasing frequency would in no way advance the completion of the particular operation. Indeed, such modification would result in a different operation, thereby, defeating the purpose of the trace. Therefore, the Office Action's asserted motivation for modifying Patel with Johnson does not apply.

Furthermore, even if the head of the trace could be more frequently accessed and modified, the Office Action has provided no explanation of how such would improve the fetch rates of the trace cache of Patel.

Moreover, the Office Action asserts that it would be obvious to modify the instruction segment of Patel with the teaching of Johnson in order to store instructions of an instruction trace in reverse order "so that the frequently accessed and modified head of the trace will be moved and modified fewer times so that performance is improved." See Office Action dated 6/12/2008, paragraph 30. Applicants respectfully disagree. Applicants submit in order to establish prima facie obviousness, there must be some suggestion or motivation to modify the reference or combine the reference teachings. For at least the following reasons, there is no such suggestion or motivation here.

As stated previously, there is no motivation to reverse the instructions in a Patel trace. Patel discloses using branch promotion to improve cache fetch rates. The purpose of branch promotion is to reduce the dynamic branching of strongly biased traces by applying static branches (or predictions). Applicants submit reversing the instructions so that the first instruction is listed last in the trace does not improve the branch promotion technique. Reversing the instructions does not reduce the dynamic branching of strongly biased traces. Moreover, it

does not improve the fetch rates. As such, there is no reason a person of ordinary skill in the art would be motivated to reverse the instructions in a trace, while using branch promotion, to improve fetch rates.

Patel also discloses using trace packing to improve cache fetch rates. The purpose of trace packing is to increase the number of instructions fetched per fetch cycle. However, Applicants submit reversing the instructions so that the first instruction is listed last in the trace does not improve the trace packing technique. Moreover, such does not appear to improve the fetch rates. As such, a person of ordinary skill in the art would not be motivated to reverse the instructions in a trace, while using trace packing, to improve fetch rates.

Applicants further respectfully disagree with the Office Action's assertion that Johnson has taught that the static sorting algorithm stores elements in reverse order, since it stands to reason that the elements first added to the array would be accessed and used before the elements most recently added to the array. *See* Office Action dated 6/12/2008, paragraph 30. The Office Action cites to column 4, lines 13-24. Column 4, lines 13-24 of Johnson state:

> For the illustrated embodiment discussed below, a static sorting algorithm is used which arranges the logical blocks of data in a logical page in reverse order, thereby placing the last logical block at the beginning of the array, and the first logical block at the end. Consequently, *the more frequent modifications* to the data in the first logical block require recompression and/or moving of fewer, or no other, subsequent frames than the *less frequent modifications* to the data in the last logical block of a page. In general, this results in a lower average number of *updated frames per data modification*, and thus improved overall performance. (*emphasis added*)

The cited section discusses improving overall performance based upon lowering the average number of update frames *per modification* in a data array. The cited section does not, however, discuss improving overall performance during *accessing* or *using* data in a data array. *Accessing* data or *using* data is not the same as *modifying* data, and the cited section of Johnson does not

discuss the benefits of its sorting algorithm during data access or data "use" at all. To assert that "modifying" data, allegedly necessitates and therefore is the equivalent of "accessing" or "using" data is to eviscerate the meaning of concepts and functions readily known to those of ordinary skill in the art as being separate and distinct.

Applicants maintain that a person of ordinary skill in the art would not be motivated to reverse the instructions in a trace, while using trace packing, to improve fetch rates and that as such, claim 1 is allowable in its present form. Independent claims 5, 6, 8 and 14 contain similar allowable limitations, and are therefore allowable for similar reasons. Claims 2-4, 7, 9-13 and 15-19 are allowable for depending from allowable base claims.

The deficiencies are not corrected by Peled. Peled discloses a cache organized around trace segments of the running programs rather than an organization based on memory addresses. However, Peled fails to provide any motivation for modifying Patel with Johnson. Moreover, there is no motivation disclosed to modify Patel with Johnson and Peled to arrive at the claimed invention. Accordingly, the Office Action has failed to establish a *prima facie* case of obviousness over Patel in view of Johnson in further view of Peled.

For at least these reasons, the Claims 1-19 are believed to be patentable over the cited references, individually and in combination. Withdrawal of the rejections is, therefore, respectfully requested.

As the cited references fail to teach or suggest each and every limitation of claim 1, they fail to support proper § 102 or § 103 rejections. Applicants submit claim 1 is allowable. Independent claims 5, 6, 8 and 14 contain similar allowable limitations, and are therefore

allowable for similar reasons. Claims 2-4, 7, 9-13 and 15-19 are allowable for depending from allowable base claims.

For at least all the above reasons, the Applicant respectfully submits that this application is in condition for allowance. A Notice of Allowance is earnestly solicited.

The Examiner is invited to contact the undersigned at (408) 975-7500 to discuss any matter concerning this application.

The Office is hereby authorized to charge any additional fees or credit any overpayments under 37 C.F.R. §1.16 or §1.17 to Deposit Account No. **11-0600**.

Respectfully submitted,

KENYON & KENYON LLP

Date: August 14, 2008

By: /Sumit Bhattacharya/_____
Sumit Bhattacharya
(Reg. No. 51,469)
Attorneys for Intel Corporation

KENYON & KENYON LLP
333 West San Carlos St., Suite 600
San Jose, CA 95110

Telephone: (408) 975-7500
Facsimile: (408) 975-7501